Management of Mobile Agent Systems: Learning From the Ants

Tony White[1], Bernard Pagurek[2], Dwight Deugo[1]
[1]School of Computer Science, Carleton University
[2]Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, ON K1S 5B6 Canada
{arpwhite@scs.carleton.ca, bernie@sce.carleton.ca, deugo@scs.carleton.ca}

Abstract

The management of mobile agent systems that solve problems in a network is an issue that must be addressed if mobile agents are to be deployed industrially. While centralized solutions are possible, where agent information is maintained on well-known servers, this is hardly desirable when distributed problem solving is one of the motivating reasons for employing mobile agents. It is clear that insufficient or excessive numbers of agents can cause the problem solving capabilities of an agent-based system to be impaired. Also, agents being software entities are almost always flawed therefore requiring the upgrade problem to be solved. This paper presents distributed algorithms based upon ant social behaviour that solve the problems of agent density maintenance and the agent upgrade problem.

Keywords: mobile agents, self-organizing agents, network management, agent upgrade

1. Introduction

This paper concerns itself with the management of software agents moving throughout a network for the purpose of solving problems using distributed computation. In any system that supports distributed computation in an unreliable network, there is a need to address issues of agent density and upgrading. The application of mobile agents, specifically netlets [1], to Network Management [2] requires that these problems be solved in order to avoid the issues described in [12]; namely, that the solution becomes a problem itself.

An excessive number of agents in a network can significantly degrade the functioning of that network, while too few may also compromise performance [2], [11]. Generally, it is sufficient to maintain agent density within a range; a single point value is unnecessarily restrictive. In fact, point control is often the cause of oscillatory behaviour in a controlled system. In an unreliable network we find that links or network nodes may fail with a resulting loss of any agents executing on the node or in transit between two nodes. Some might argue that the computing infrastructure should support reliable computation and transport. However, this adds significantly to the complexity of the mobile agent infrastructure required and seems unnecessary when we observe that naturally occurring agent systems (for example ants and wasps) are extremely tolerant to individual agent loss. Maintaining accurate statistics on agent numbers and position is really unnecessary for solving this problem if we view it from a decentralized viewpoint. In fact, agent populations should self-organize, as is clearly seen in nature [17], [18].

It is also too commonly the case that software, and agents are not likely to be exceptional here, is flawed either logically or functionally. When software flaws are discovered, software modification of individual agents or complete replacement of the defective agent must occur. A computing infrastructure supporting agent versioning is a more challenging problem than density maintenance, in that we know neither the positions of individual agents nor the versions actively moving through the network. This presents a serious halting problem in that we do not know the numbers of particular versions of the agent that are active in the network. It is not possible, then, to know when the upgrade process is complete. In other words, any algorithm or solution technique should be capable of upgrading older versions of agents for all time.

It is difficult to conceive of an environment that automatically upgrades agent software when changes are available that does not rely on some form of global information. For example, the current mechanism for software upgrade used on personal computers is to check periodically with the supplier of the software and download improvements when available. While mirror sites partially solve the load balancing problem, the solution is still a central one, one in which global information is held on the supplier's web site. Should the supplier move or disappear completely, the upgrade process fails. This, obviously, is an inferior solution. A decentralized solution to the versioning problem would have no such limitation, relying instead on only local information.

This paper proposes the use of algorithms that exploit ideas inspired by ants; relying exclusively on local information and the emergent behavior of large numbers of agents. The paper consists of four further sections. Section 2 provides a brief description of the motivating ideas. Section 3 provides a description of the agent density control problem and how it can be solved using stigmergic communication. Section 4 addresses the problem of upgrading software agents in a network and provides algorithms for the upgrade problem. The paper concludes with a section that summarizes the key contributions of the paper.

2. Motivating Ideas

It is difficult to argue against the effectiveness of many naturally occurring multi-agent systems and, in particular, systems exhibiting mobility. Societies of simple agents are capable of complex problem solving while possessing limited individual abilities [17]. Many algorithms inspired by the social behaviour of insects have recently been documented [18] with Dorigo being acknowledged as having introduced ant-based search [5].

Problem solving by societies of simple agents has a number of common characteristics. Inter-agent communication is local; no single agent has a global view of the world. Communication is also achieved using simple signals and these signals dissipate with time. Signal levels provide the driving force for migration patterns. Individual agents sense and contribute signal energy to the environment. In this description of the problem solving process, there are two distinct and important agent characteristics. First, there is the role of the agent within the problem solving process; i.e., how the work of problem solving is distributed to a *diverse* set of agents. Second, the degree to which the actions of one agent reinforce the actions of other agents in the society of problem solvers is significant. The appeal of swarms of biologically inspired agents for industrial problem solving has recently been appreciated [4]. Research into the problems and potential of multiple, interacting swarms of mobile agents is just beginning [3].

Appealing to Grassè's principle of stigmergy (see [18]), ant-inspired agents solve problems by moving over the nodes and links in a network and interacting with "chemical messages" deposited in that network. Chemical messages have two attributes, a *label* and a *concentration*. Chemical messages are used for communication rather than raw operational measurements from the network in order to provide a clean separation of measurement from reasoning. In addition, chemical messages drive the migration patterns of agents, the messages being intended to lead agents to areas of the network that may require attention. Chemical labels are digitally encoded, having an associated string pattern that uses the alphabet {1, 0, #}. This encoding has been inspired by those used in Genetic Algorithms. The hash symbol in the alphabet allows for matching of both one and zero and is, therefore, the "don't care" symbol.

3. Density Control

The problem of resource control for mobile agents can be attributed to Tschudin [8] and has been studied by several researchers [9], [10], [11], [13], [14], [15] and [16]. Shehory et al [11], for example, uses agent cloning to ensure appropriate agent densities for problem solving while acknowledging the relevance of the load balancing literature, while Bredin et al [10], [14] uses Market Based Control for resource allocation problem resolution. Clearly, mobile agent researchers have long recognized the importance of having an appropriate number of agents in a network performing a given task. For example, in the routing problem described in [7], it was noted that if too few routing agents were sent out into the network, routes would not necessarily emerge. This is in complete agreement with Dorigo's work on Ant Search [5], ant-based problem solving and AntNet [18].

The above scenario is a less interesting example of density control when compared to the general situation with agents moving through the network and never terminating their problem solving activity; for example fault detection using netlets [1], [2]. It is this type of problem that motivates the research reported here.

Consider, then, the problem of multiple swarms of problem solving agents in an unreliable network. Clearly, if steps are not taken to inject new swarm agents over time, agent density will tend to zero. The argument is straightforward. Assuming that agent movement is random and uncorrelated, given a non-zero component failure rate, λ_f, following a Poisson distribution, a network of n nodes, with m agents, the number of agents failing per unit time is: $m\lambda_f/n$. This expression, being unconditionally greater than zero given n,m greater than zero, ensures that the probability that the number of agents in the network at time t_1 is less than the number of agents in the network at time t_0, $t_1 > t_0$ is one.

Having established the need for agent replacement, a mechanism for that replacement is required. We propose the addition of a Density Control Agent (DCA) class. The purpose of the density control agent class is to circulate continuously in the network depositing chemical signals in that network such that agent classes whose density is being controlled will automatically adjust their numbers to fall within the target density range. The DCA class uses a random migration decision function in order to explore all parts of the network equally and is responsible for controlling its own density. It also remains at each node for a randomly generated period chosen from a uniform distribution in order to avoid correlations between agent actions. This is an important observation as, without it, significantly greater oscillations are observed with possible population extinction. Every agent class that is density controlled generates a visit chemical that is sensed by the DCA. The DCA controls its own density in order to solve the problem of managing management class agents. Therefore, the DCA also generates a visit chemical. Visit chemical concentrations are associated with the node. The visit chemical leaves a trail of activity for the density-controlled problem solving agents that is integrated across a number of network nodes by DCA agents for the purpose of generating birth or death signals that are in turn sensed by the density controlled problem solving agents. Birth or death signals are, naturally, chemical in nature and these chemicals do not evaporate.

DCA agents generate birth signals when the aggregated visit chemical concentration for a particular density control problem solving agent class falls below a threshold value. Visit chemical concentrations evaporate over time, this forming the dissipative field that makes the density control mechanism work. This is a crucial part of the control process, as, without it, visit chemicals would accumulate forever leading to the rapid extinction of the entire agent population. In fact, we make use of this observation in solving the agent upgrade problem described in Section 4. DCA agents generate death signals when the aggregated visit chemical concentration for a particular density control problem solving agent class exceeds a threshold value. Both types of signals are generated on the node where the appropriate threshold condition is violated. An exponential averaging process is used to aggregate visit chemical concentrations.

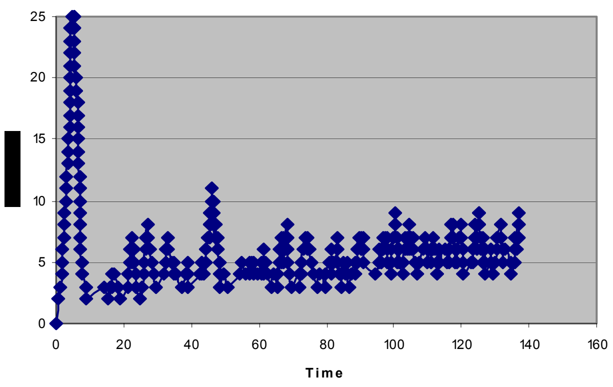
When a density controlled problem solving agent senses a birth signal, it clones itself, generating a new agent with the parent agent consuming the birth signal. When a density controlled problem solving agent senses a death signal, it chooses to die according to a probability distribution, having first consumed the death signal.

3.1 Results

In order to demonstrate the utility of the above algorithm, the two networks used for routing experiments [7] were revisited for density maintenance. Two classes of agent, including the DCA, both with random migration decision functions, were allowed to circulate within the network. A single agent of each class was injected into the network and allowed to stabilize to the “natural” value for the network. In later experiments, extra agents were injected into the network periodically in order to see if the density correction algorithm could return the density to the appropriate value for the network. This had the added side effect of ensuring that at least one DCA agent would be present in the network.

The minimum concentration threshold value was set at 1, the maximum at 5. The rate at which visit chemical was deposited on the node was 1.5 units per visit; the evaporation rate was set to 0.8 units per simulation time step. Figure 1 shows the variation of agent number with time for graph1 used in the routing experiments previously described [7]. The agent type plotted represents a very simple agent that has a random migration pattern and is designed to measure the concentration of visit chemical, nothing more. The focus, in this study, is the management of agent number and not problem solving per se. The variation of agent number with time represents the self-regulation of agents in the system; no further agents beyond the initial seed agent were injected into the network. The rapid rise in agent number initially is due to the fact that the network contains no visit chemical traces for the agent type. As a consequence of this, birth signals will be generated for all nodes visited and a large number of new agents will be generated. Avoiding this transient is possible by injection of the DCA *after* the network has stabilized, i.e., after the problem solving agents have had time to colonize the network and lay down visit chemical traces that mark their presence in the network.

Even with the start up transient, the network quickly recovers and settles down to a number of agents that oscillates around 5. Similar behavior can be observed in Figure 2, where agent density control is applied to graph2 [7]. In the experiment shown in this figure, the agent number oscillates around 7, with the number of agents never falling below 5. The oscillation can be further damped by choice of exponential averaging constant and by adjustment of the minimum to maximum visit chemical concentration threshold ratio. In the experiments charted in Figure 2 and Figure 3, a ratio of 5 was used to control the agent density.



[redacted]

Figure 1. Agent Number vs Time

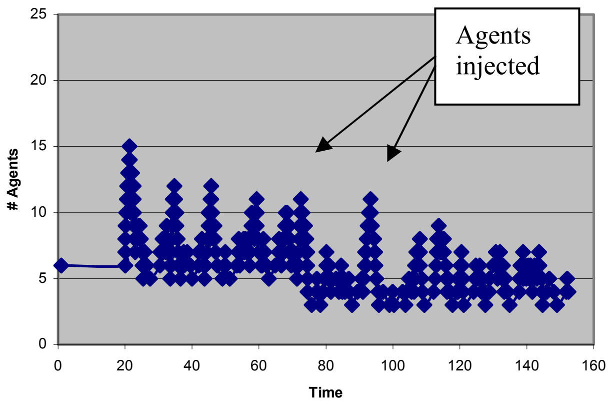


Figure 2. Agent Number vs Time

Figure 2 demonstrates the utility of having density control come online once the network has been colonized by problem solving agents. In these experiments, density control was disabled during the first 20 time units of the simulation. Interestingly, Figure 2 shows the system moving from one stable state to another at approximately 72 time units. Continuing the simulation beyond 200 time units saw no further changes in state. Contrasting the dynamics of this system with those displayed in Figure 2 clearly shows a much smaller transient and more rapid stabilization to the steady state network behaviour. The number of agents injected initially seems to make minor differences to the final stable network state. Experiments were run wherein up to 30 agents were injected initially; the system still converged to a mean number of agents of 6.

Experiments were conducted where agents were occasionally injected into the network in order to test the stability of the agent density management algorithm. As Figure 2 shows, injecting agents after the network (at 50 time units) has settled down merely causes the agent density to find a new stable point, which may, of course, be the same as the original point. This is to be expected in that many systems have several basins of stability. This characteristic is an attractive feature of the system in that we can alter the stable system trajectory by injection (or removal) of agents. In fact, as suggested earlier, we would propose the periodic injection of a density control agent in order to ensure that the system never remains locked at zero population for that agent. Figure 3 also shows the destruction of agents in the network after the settling period. This scenario represents the situation that initially inspired the density management algorithm, namely the loss of agents as a consequence of network component failure. Figure 3 shows two failures, at approximately 140 and 175 time units, where multiple agents are lost. Clearly, the algorithm has performed well, with the natural trajectory of the system being quickly restored. Obviously a failure of *all* components in the network would cause the loss of all agents; however, the scenario demonstrated in Figure 3 actually represents a failure of 25% of the network that, in all likelihood, represents an extreme case. More general experiments, with random single node failures, provided equivalent support for the robustness of the density control algorithm and results are not included here as, it was felt, the scenario described in the previous paragraph provides a more dramatic illustration of the robustness of the algorithm.

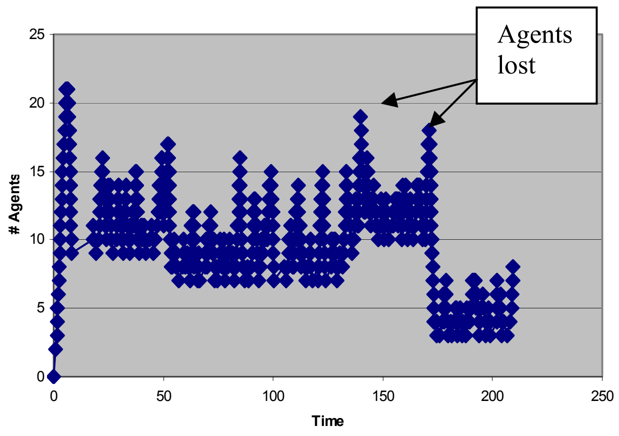


Figure 3. Agent Number vs Time

4. Agent Upgrading

Software rarely has completely correct behaviour when first deployed. The problem of upgrading software in an operational environment is challenging and is currently the focus of considerable research [19], [20] and [6]. Hofmeister [19] describes three forms of dynamic reconfiguration: module replacement, structural change and geometric replacement. The software upgrade problem presents a unique challenge when the software is an agent and that agent is mobile, as we have no knowledge a priori of the location of any agent. The software upgrade problem is further complicated by no knowledge of the number of agents to be upgraded and their source of injection into the network. This latter piece of information is important as it implies that older, incorrect versions of a software agent may be injected into the network once the upgrade process has been supposedly completed. Together, these problems present a significant research problem, making geometric replacement the most attractive mechanism for upgrading agents. Again appealing to ant-like problem solving agents, and their tendency to be robust with respect to the failure of an individual agent, we view the agent upgrade problem as one of "failing" the faulty agent and injecting one with corrected behaviour.

Not knowing the source of the agent originally injected into the network leads to the disturbing and inevitable conclusion that the upgrade process is potentially never complete as an obsolete agent may be introduced from a source that has yet to receive the corrected agent.

Referring now to the previous section on density control, a potential mechanism for removal of the faulty agent is to exploit its response to the death signal and visit chemical concentrations. That is, increase the levels of the visit chemical for the faulty agent such that an associated DCA agent generates the death signal for that agent throughout the network. This is achieved by having different visit chemicals for the faulty and corrected agents with the encodings so chosen that the faulty agent senses the visit chemical of the correct agent. Hence, fault agents will tend to see higher concentrations of visit chemical when compared to the corrected agent; the corrected agent will tend only to see its own. An example best illustrates this.

Consider two agent versions, v_f and v_c, representing faulty and corrected versions respectively having visit chemical encodings ####1 and ###11 respectively. Assuming a string length of 5, if we were to have concentrations c_f and c_c of the two chemicals, the faulty agent would sense $c_f + c_c$, whereas the corrected agent would see c_c. In other words, the faulty agent would see higher concentrations of visit chemicals and would be more likely to see the death signal as a result of exceeding the upper bound on visit chemical concentration. Similarly, the faulty agent would be less likely to see the birth signal as a result of higher visit chemical concentration.

This example can easily be extended. Consider a perfect agent that corrects faults in agent version v_c, say v_p. Let v_p have the visit chemical encoding ##111 and concentration c_p. Again assuming the Binary Array Chemistry of order 5, if we were to have concentrations, the faulty agent would sense $c_f + c_c + c_p$, whereas the corrected agent would see $c_c + c_p$. In this case, both v_f and v_c would experience a reduced number of birth signals and elevated number of death signals. As the number of v_p agents increases, the tendency is for the number of v_f and v_c agents to decrease, eventually causing the imperfect agent types to disappear.

This mechanism works because of the encoding scheme used for the three agent types. It relies upon the masking of the existence of previous versions of the agent through the increasing number of bits being specified as we move to higher and higher versions of the agent. Obviously, this limits the number of versions that might be accommodated. However, a potentially infinite data structure such as can be provided by a tree removes this limitation. Obviously this is a more expensive solution from a computational viewpoint -- pattern matching by bit position versus matching by subtree -- but does solve the more general problem when an unbounded number of agent versions needs to be supported.

4.1 Agent Upgrading Results

This section presents experimental results that demonstrate the applicability of the agent upgrade algorithm. In the results presented below, the two networks used in the density control experiments were also used as a simulation test bed for the agent upgrade algorithm.

In Figure 4, three flawed agents are injected into the network and are subject to the density control algorithm. Eventually the number of agents stabilizes around 7. At 83 time units, a single corrected agent is injected into the network and quickly establishes dominance over the flawed agent using the visit chemical masking mechanism introduced in the previous section. It is not possible for the population of flawed agents to recover once the number of flawed agents reaches zero, as the birth process is one of cloning. Only through injection of a new flawed agent into the network from some external source can the flawed population temporarily recover.

In Figure 4, the flawed agent type is re-injected into the network at 140 time units following the colonization of the network by the corrected agent. The flawed agent is quickly removed from the network on both occasions despite large numbers of them being injected. While results are not included here, the number of flawed agents re-injected did not affect the final state of the network, merely the time to achieve it. In all cases, the flawed agents were eventually removed from the network, leaving only

Figure 4. Agent Number vs Time

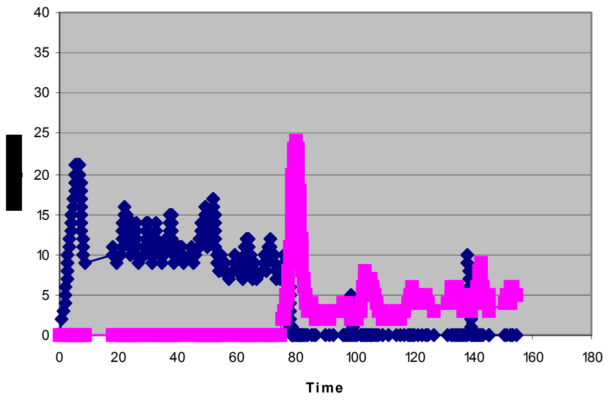
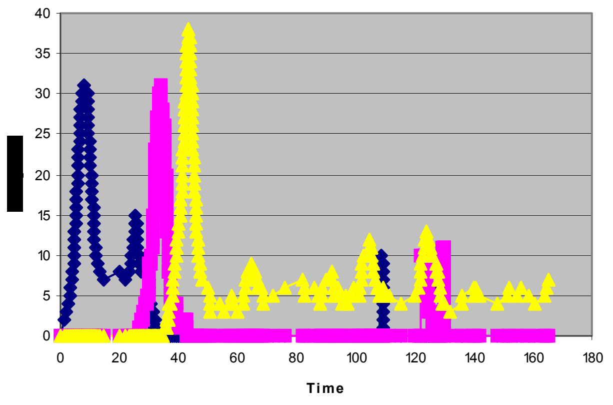


Figure 5. Agent Number vs Time



the corrected agent population. Clearly, this demonstrates that the corrected agent resists recolonization of the network by the flawed agent. The rejection of the flawed agents by the network is faster upon re-injection as a result of the established population of corrected agents.

In order to demonstrate the robustness of the algorithm in the presence of multiple agent versions, experiments were conducted wherein a third version -- the "perfected agent" -- was injected into the network after the corrected agent population had replaced the initial population of flawed agents. As can be seen in Figure 5, the perfected agent quickly established a dominant position in the network causing the corrected agent population to vanish completely. Once again, subsequent re-injection of flawed or corrected agent populations did not affect the dominant position of the perfected agent. This is clearly shown in Figure 5 where corrected and flawed agents are re-injected into the network following the perfected agent establishing itself in the network. Reviewing the interval 25 to 45 time units in Figure 5 also shows that the algorithm can deal with multiple agent populations simultaneously trying to establish themselves. During this time interval, flawed, corrected and perfected agent types are moving throughout the network; however, the corrected agent population is quickly extinguished even before stabilization can occur. Despite the version control algorithm's simplicity, it seems to be remarkably effective in maintaining the correct dominant version in the network. While the above results have demonstrated the effectiveness of the upgrade algorithm, a comment regarding the rate of production of visit chemicals for successive versions of the agent needs to be made. As stated previously, given two agent versions, v_f and v_c, with encodings as previously described, v_f will see higher concentrations of visit chemical when compared to v_c. If the rates of production per nodal access of visit chemical are r_f and r_c for faulty and corrected agents respectively, and the visit chemical sensitivity ranges are (l_f,u_f) and (l_c,u_c) respectively, we can accelerate the removal of faulty agents by setting the corrected agent visit chemical production rate such that $r_c > u_f$. If this relation holds, the faulty agent will tend to see quantities of visit chemical that exceed its upper density bound, resulting in the generation of the death signal for the faulty agent class in almost all instances. The word almost applies here, as we have to allow for the effects of evaporation in our system. Between the departure of a corrected agent and the arrival of a faulty agent evaporation can reduce the concentration of the visit chemicals sensed by the faulty agent below u_f. In order for the corrected agent population to maintain density in a similar way to the flawed agent population with the relationship $r_c = u_f$ holding other parameters have to be modified too. Assuming the evaporation rate is constant, the following have to hold: $u_c = u_f r_c/r_f$ and $l_c = l_f r_c/r_f$..

5. Conclusions

In a network managed completely by swarms of mobile agents, two important observations need to be made. First, the number of agents is unknown and second, the positions of mobile agents are unknown. These characteristics make management of the swarm populations extremely challenging. This paper has addressed two questions related to the management of mobile agent swarms.

The first question relates to the maintenance of population density and we have presented algorithms that maintain population density in a network having unreliable components. The algorithms presented rely on local knowledge only and we have shown by experiment that populations quickly settle to a mean population around which they oscillate. The algorithms presented are robust with respect to the introduction of agents after the network has stabilized as well as loss due to component failure.

The second question deals with the upgrading of agents over time. Agents, being software entities, rarely have correct behaviour when first introduced into service and often require upgrades. This paper presents an algorithm that solves the upgrade problem by taking advantage of the density control algorithm in a form of parasitic behaviour where a corrected agent "fools" the flawed agent population into believing that there are more of them than there are, the flawed agent population quickly decaying to zero. We believe that these algorithms, once again, clearly demonstrate the value of "learning from the ant".

6. References

[1] Bieszczad A. and Pagurek, B., Network Management Application-Oriented Taxonomy of Mobile Code, Proceedings of the IEEE/IFIP Network Operations and Management Symposium NOMS '98, New Orleans, Louisiana, February 1998.

[2] Bieszczad A., White, T., Pagurek, B., Mobile Agents for Network Management. In IEEE Communications Surveys, September 1998.

[3] White T., SynthECA, A Synthetic Ecology of Chemical Agents, Ph.D. thesis, Carleton University, 2000.

[4] Parunak H. V. D., Go to the Ant: Engineering Principles from Naturally Multi-Agent Systems, Annals of Operations Research, 75:69-101, 1997.

[5] Dorigo M., Maniezzo V. and Colorni A., The Ant System: An Autocatalytic Optimizing Process. Technical Report No. 91-016, Politecnico di Milano, Italy, 1991.

[6] Ning F., Software Hot Swapping, Master of Engineering thesis, Department of Systems and Computer Engineering, Carleton University, September 1999.

[7] White T., Pagurek B. and Oppacher F., Connection Management using Adaptive Mobile Agents, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98), July 1998.

[8] C. Tschudin. Open Resource Allocation for Mobile Code. *Mobile Agents - First International Workshop, MA '97* (Berlin, Germany, April 7-8, 1997). Published as Kurth Rothermel and Radu Popescu-Zeletin, editors, *Lecture Notes in Computer Science*, **1219**, Springer, 1997.

[9] Jonathan Bredin, Rajiv T Maheswaran, Cagri Imer, Tamer Basar, David Kotz, and Daniela Rus; A Game-Theoretic Formulation of Multi-Agent Resource Allocation, Proceedings of the 2000 International Conference on Autonomous Agents, Barcelona, Spain, June 2000.

[10] Jonathan Bredin, David Kotz, Daniela Rus; Economic Markets as a Means of Open Mobile-Agent Systems, Appears in the Proceedings of the Workshop of Mobile Agents in the Context of Competition and Cooperation as part of the Third International Conference on Autonomous Agents, Seattle, WA, May 1999.

[11] O. Shehory, K. Sycara P. Chalasani and S. Jha Agent cloning: an approach to agent mobility and resource allocation**,** IEEE Communications, pages 58-67, vol. 36 no. 7, July 1998.

[12] Simoes P., Moura e Silva L., and Boavida F., Mobile Agent Infrastructures: a Solution for Management or a problem to Manage, in Proceedings of the 3rd IEEE Conference on Telecommunications, 23-24 April 2001, Figueira da Foz, Portugal.

[13] Mudumbai Ranganathan, Anurag Acharya, and Joel Saltz. Distributed resource monitors for mobile objects. In Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems, pages 19-23, Seattle, Wa., October 1996.

[14] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based resource control for mobile agents. In Proceedings of Autonomous Agents '98, pages 197-204, 1998.

[15] Jonathan Bredin, David Kotz, and Daniela Rus. Utility driven mobile-agent scheduling. Technical Report PCS-TR98-331, Dept. of Computer Science, Dartmouth College, May 1998.

[16] L. Yamamoto and G. Leduc. An Agent-Inspired Active Network Resource Trading Model Applied to Congestion Control. In Proceedings of the MATA 2000 Workshop, Springer LNCS 1931, pages 151--169, Paris, France, September 2000.

[17] Hölldobler B. and Wilson E.O., Journey to the Ants. Bellknap Press/Harvard University Press, 1994.

[18] Bonabeau E. Dorigo M. and Theraulaz G, Swarm Intelligence, Oxford Press, 1999

[19] Hofmeister C. Dynamic Reconfiguration of Distributed Applications, Ph.D. thesis, Computer Science Department, University of Maryland, 1993.

[20] Noel de Palma, Dynamic reconfiguration of agent-based applications, Technical Report Project SIRAC, INRIA, 1999.